Does it Pay Off? Model-Based Verification and Validation of Embedded Systems!

Frits Vaandrager

Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
F.Vaandrager@cs.ru.nl

Abstract. An overview is presented of the state-of-the-art in model-based verification and validation of embedded systems, directed towards an industrial audience. Verification and validation consists in exploring the current design against properties expressed as part of the requirements. It includes testing, model checking, runtime verification and fault-diagnosis, and more exploratory techniques such as the use of theorem proving. During recent years, much progress has been made in theory, methods and tools for model-based verification and validation. In this paper, I will try to indicate for what type of practical problems it pays off to apply one of these modern techniques. Special attention will be paid to the results of six PROGRESS projects in this area.

1 Introduction

Embedded systems are highly specializable, often reactive, sub systems that provide, unnoticed by the user, information processing and control tasks to their embedding system. Embedded systems are omnipresent nowadays and make possible the creation of systems with a functionality that cannot be provided by human beings. Example application areas are consumer electronic products (*e.g.* CD players, microwave ovens), telecommunication (*e.g.* mobile phones), medical systems (*e.g.* pacemakers), traffic control (*e.g.* intelligent traffic lights), driving and car control (*e.g.* ABS), airborne equipment (*e.g.* fly-by-wire), and plant control (*e.g.* packaging machines, wafer steppers). The term embedded system thus encompasses a broad class of systems, ranging from simple microcontrollers to large and complex multi-processor and distributed systems. The huge economic importance of embedded systems is undisputed.

Some characteristics of embedded systems are:

- Complex interaction with the environment. Embedded systems can only be designed and analyzed if one takes the behavior of their environment into account. Frequently this environment is highly nondeterministic and intrinsically continuous.
- A multitude of *quantitative* constraints. These constraints involve the resources that a system may use (computation resources, power consumption,

memory usage, communication bandwidth, costs,..), assumptions about the environment in which it operates (arrival rates, hybrid behavior), and requirements on the services that the system has to provide.
- High dependability requirements. Besides functional constraints many other aspects play a role in the design of embedded systems: timeliness, fault tolerance, availability, security, safety, *etc.*.
- Design and manufacturing costs are very important.

This combination of factors makes the design of embedded systems in general a very complex task. Failure of embedded systems often may have serious consequences (loss of lives, huge financial losses), so correctness and reliability are of vital importance. As a result it is common for more than 75% of embedded software development costs to go into validation and verification. So there is a lot of potential for saving money.

Validation and Verification There is quite some confusion in the literature about the meaning of the terms validation and verification. I prefer to remain consistent with the traditional usage of these terms [4, 21]. *Validation* aims at increasing confidence in the correct operation of an implementation. Are we building the right system? Ideally, the desired behavior of a system is fully specified in advance, but in practice it rarely occurs that we know exactly how a system should behave under all possible circumstances. There exist two basic validation strategies, viz. the verification strategy and the falsification strategy. The objective of *verification* is to show that an implementation possesses a property prescribed by its specification. Are we building the system right? An implementation is considered correct (or valid) if all properties prescribed by the specification are present in the implementation. In *falsification* the objective is to try and show that the negation of a specification requirement holds in an implementation. In this case an implementation is considered correct if all attempts to falsify a requirement fail. Note that therefore in verification an implementation is rejected as a correct implementation if it does not posses a prescribed property, whereas in falsification an implementation is rejected when it does posses the negation of a prescribed property. Ideally, verification and falsification are complementary notions in the sense that "falsification equals verification of the negation". However, in practice falsification is much weaker than verification (see *e.g.* Popper [29]). Both verification and falsification can be used simultaneously for assessing the correctness of implementations.

Models provide (mathematical) abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on relevant ones. All forms of engineering rely on models to understand complex, real-world systems. Models may be developed as a precursor to implementing the physical system, or they may be derived from an existing system or a system in development as an aid to understanding its behavior. In the software engineering world, modeling has a rich tradition, dating back to the earliest days of programming. Boosted by the work of the Object Management Group (OMG) on

the Unified Modeling Language (UML) and Model Driven Architecture (MDA), the role of models during application design, implementation, verification and validation has become much more important in recent years, and this is a very positive development. Model-Driven Development (MDD) is a system development technique in which the primary artifact is a *model*, which is a collection of views. Ideally, the technique allows engineers to (graphically) model the requirements, behavior and functionality of computer based systems. The model allows all the stakeholders to participate in the development process. The design is iteratively analyzed, validated, and tested throughout the development proces while automatically generated production quality code can be output in a variety of languages.

The promise of model driven development is to allow definition of machine readable application and data models which allow long-term flexibility of:

- *implementation:* different/new implementation infrastructure can be integrated or targeted by existing designs.
- *integration and component reuse:* since not only the implementation but the design exists at time of integration, we can automate the production of data integration bridges and the connection to new integration infrastructures. The availability of interface models of components facilitates reuse.
- *verification and validation:* since the developed models can be used to generate code, they can equally be validated against requirements, tested against various infrastructures, and can used to directly simulate the behavior of the system being designed. *Formal verification* is the process of mathematically checking that the behavior of a system (component), described using a formal model, satisfies a given property, also described using a formal model.
- *maintenance:* the availability of the design in a machine-readable form gives developers direct access to the specification of the system, making maintenance much simpler

Depending on the role that models play in the design process, we see different types of models. Ideally, there are well-defined relationships between these models. Models from which *implementations* can be generated are typically constructed using commercial tools such as Rational Rose, Rhapsody and visualSTATE, and usually contain a lot of details, including code fragments. *Interface models* focus on the external behavior of systems and components. These models typically are much more abstract. A useful classification of component specifications ("*contracts*") has been proposed by Beugnard et al [3], where a hierarchy is defined consisting of four levels:

- *Level 1:* Syntactic interface, or signature (*i.e.* types, fields, methods, signals, ports, *etc.* that constitute the interface).
- *Level 2:* Constraints on values of parameters and of persistent state variables, expressed *e.g.*, by pre- and post-conditions and invariants.
- *Level 3:* Synchronization between different services and method calls (*e.g.*, expressed as constraints on their temporal ordering via state machines or temporal logic).

– *Level 4:* Extra-functional properties (in particular real-time attributes, performance, QoS (*i.e.* constraints on response times, throughput, *etc.*.) This level can be separated into two aspects
  4a timing properties (*e.g.* absolute time bounds)
  4b Quality of Service properties, typically given by performance measures, often formulated in stochastic terms (*e.g.* average response time).

Currently, most component modeling frameworks support only level 1 contracts, while some also support level 2 and 3 contracts (for instance, the ISpec framework studied in PROGRESS project EES.5141). For embedded systems Level 4 properties are important (their specification has been studied in PROGRESS project TES.4999). Models for *verification and validation*, finally, typically focus on some specific aspects of a systems behavior. They are as abstract as possible in order to make analysis tractable.

A serious practical question is how much effort to put in building models. Constructing good models is difficult and most software developers dislike writing them. Aiming at the highest quality models, *e.g.* by using formal specification techniques, is expensive and requires highly skilled developers. Spending minimal effort on constructing models can also be expensive except that the costs are incurred later in the systems life cycle by increased maintenance costs, customer dissatisfaction, *etc.*. In practice the right balance between these two extremes has to be found. In finding the balance several issues have to be taken into account such as the expected lifetime and usage of the models, the skills of the readers and writers of the specifications, how critical the interfaces are, *etc.* [24].

Simulation remains the main tool to validate models, but the importance of formal validation & verification is growing, especially for safety-critical embedded systems. Although still in its infancy, it shows more promise that verification of arbitrary systems, such as generic software programs, because embedded systems are often specified in a restricted way [16]. Simulation of embedded systems is challenging because they are heterogeneous. In particular, most contain software and hardware components that must be simulated at the same time (this is the co-simulation problem). Although there is a lot to say about simulation, I will focus in this paper on formal verification, also because this has been the main research topic of research within the PROGRESS research projects that I have been asked to discuss.

2 Formal Methods

Mathematics has always been of great importance in engineering. The term "formal methods" is commonly used to refer to the applied mathematics of computer system engineering. Whereas traditional engineering disciplines rely heavily on continuous mathematics (analysis, numerical computation), the design of dependable computer based systems requires a more discrete style of mathematical reasoning. These systems are typically modelled as discrete event dynamical systems (state machines, automata) and their specification and analysis requires the use of mathematical logic and advanced search algorithms to enable model

checking and theorem proving. But also quantitative approaches and continuous mathematics are increasingly applied in formal methods, for instance in performance analysis and design of hybrid systems. In this paper, I will focus my attention to formal methods for validation and verification.

One only has to open up any book on algorithms to see that mathematics plays a key role in their verification and analysis. Nevertheless, most software engineering projects hold formal methods at arm's length unless they involve critical systems [9, 31]. Is this due mathfobia? Is it a matter of lack of training? Or is application of formal methods simply not cost effective? In an attempt to answer these questions, I will discuss a spectrum of formal methods, ranging from from cheap and incomplete to expensive and complete (see Figure 1, adapted from Rushby).

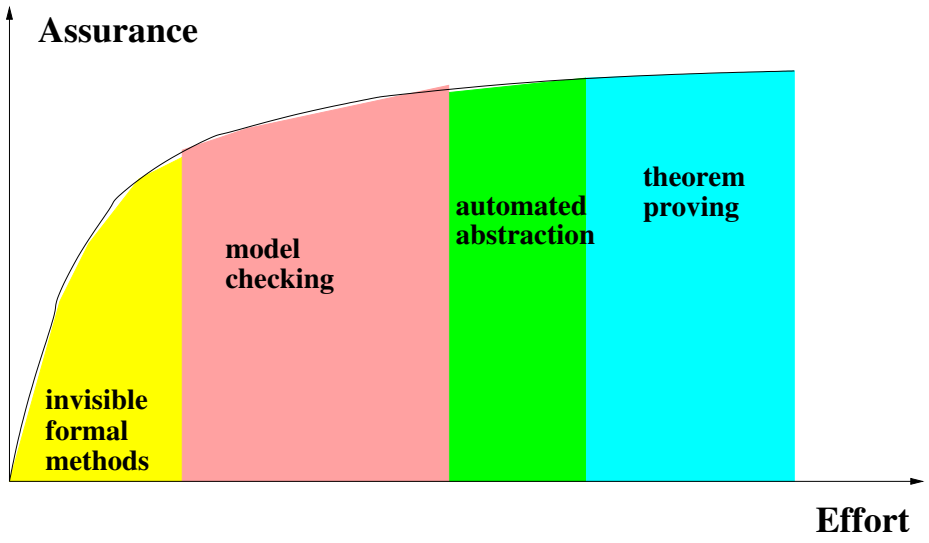

Fig. 1. A spectrum of formal methods.

Berry has suggested to use the term "automatic bug detection" in place of "formal verification" to underscore that it is too much to hope for a conclusive proof of any nontrivial design. Instead the goal of formal verification should be a technology that will help designers preventing problems in deployed systems. The point is that in most cases a formal model is a rather aggressive abstraction of the real design that it intends to capture, and similarly a formal specification is typically just an abstraction of a fragment of the full set of requirements. Therefore, if we manage to formally verify that the model satisfies its specification we may usually not conclude that the systems meets its requirements. However, a counterexample found during formal verification often leads to the discovery of a flaw in the design. In fact, formal verification turns out to be an

extremely effective method for finding bugs. The paradox is that the verification at the level of the formal model often amounts to falsification of the real system! Given the fact that people often mix up a model of a system with the system itself, this paradox has created an enormous amount of confusion.

A basic idea in formal methods is to use *symbolic calculation*. A single symbolic calculation can subsume many individual numeric cases (just as $x^2 - y^2 = (x - y) \times (x + y)$ subsumes $36 - 16 = 2 \times 10$, $49 - 4 = 5 \times 9$, *etc.*). By using symbolic calculation, formal methods tools can search huge state spaces (trillions of reachable states) efficiently. As a result, these tools can be used to find rare error scenarios as well as to verify their absence. Symbolic calculation is mechanized using the methods of automated reasoning: theorem proving, model checking, constraint solving, *etc.*. There has been sustained progress in these fields for several decades and they have recently broken though the barriers to practical application.

2.1 Theorem Provers

As soon as both a system and its specification have been modelled as mathematical entities, verification essentially amounts to proving a mathematical theorem. Following the pioneering work of N.G. de Bruijn on Automath[1], many proof assistents have been developed: software tools in which mathematical theories can be expressed and the correctness proofs of mathematical theorems can be checked mechanically and interactively, *e.g.* PVS, ACL2, HOL, Isabelle, Nuprl and Coq. Use of interactive theorem proving requires great skill and resources but allows one to solve very hard problems.

The most prominent commercial application of theorem proving has been by Intel Corporation in the area of hardware verification. Intel wrote off 475 million USD to cover damages for the incident with the incorrect division in early Pentium Processors (also known as the FDIV bug), which occurred in 1994. A similar problem in current chip designs would be much more costly. Chip designs are getting more complex, but the associated testing problem is growing even faster in size and complexity. Traditional testing techniques are not sufficiently powerful and formal verification techniques can sometimes offer a solution. In fact, since the FDIV bug formal verification has become almost standard practice in the hardware industry. Twenty percent of the Pentium IV design was formally verified and many high-quality bugs were discovered before "first silicon". The HOL light theorem prover was used by John Harrison and his team to verify the floating point operations of the Itanium processor. As was to be expected, several bugs were found in the design. The verification also increased the problem understanding, which eventually led to several improvements in the design [27].

Within Dutch universities there is extensive expertise on theorem proving. The EU IST Verificard project, that was coordinated by the University of Nijmegen, built a verification tool called LOOP on top of the theorem prover PVS

[1] See `http://automath.webhop.net/`.

to verify software for Java Card smartcards. The European smartcard industry needs this type of technology to obtain security certifications at the highest levels of the Common Criteria standard, an international standard (ISO 15408) for computer security evaluations. The project team found a coding error in a critical smartcard application, enabling continued trust and reliability in the application in question.

Within PROGRESS project CES.5009, the PVS theorem prover has been used to study transparent replication of Splice components. A problem with time stamps prevented full transparency of replication. A solution was proposed by the project (involving the copying of time stamps in certain situations) and adopted in a newer version of Splice.

Despite these success stories it is fair to say that at the moment direct application of interactive theorem proving tools is not cost-effective for Dutch industry, except possibly for a few small niche areas.[2]:

One of these niche areas might be the high-level description and analysis of architectures. The higher-order logic languages used by general purpose interactive theorem provers are extremely expressive and allow for concise description of all the concepts that play a role within an architecture (the input languages for other verification tools are typically much less expressive). Typically, the number of concepts involved is not too big, and the central role of an architecture in a design justifies a serious investement in (formal) validation en verification. Within PROGRESS case studies in this direction have been carried out by project CES.5009 (Splice) and project TES.4999 (the Æthereal network on chip [19]). Theorem provers are also an important area for academic researchers. They are widely used for verification of complex distributed and real-time algorithms, and on the long run they will revolutionize the way mathematicians work. A recent breakthrough was obtained by Benjamin Werner (INRIA) and Georges Gonthier (Microsoft Research), who succeeded in 2004 to use the Coq proof assistent to create a surveyable proof of the celebrated four color theorem.

2.2 Model Checking

Model checking is emerging as a practical tool for automated debugging of complex reactive systems such as embedded controllers and network protocols. In model checking, specifications about the system are expressed as (temporal) logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Extremely large state-spaces can often be traversed in minutes. Model checkers were initially developed to reason about the logical correctness of discrete state systems (SMV, CADP, SPIN,μCRL,SAL), but have since been extended to deal with real-time (Uppaal), probabilistic systems (PRISM) and limited forms of hybrid systems (Hytech, PHAVER).

[2] The basic algorithmic techniques and decision procedures used within theorem provers (resolution, BDDs, SAT solving,..) are applied successfully in many of the invisible formal methods, to be discussed in Section 2.4.

The two greatest advantages of model checking over theorem proving are (1) once a model and a property are specified, analysis is in principle fully automatic; (2) the ability to produce counterexamples that can be used in testing, debugging or other analysis. Disadvantages result from the tradeoffs made to make automation possible: in particular the expressivity of the modelling and specification languages is limited. Model checking tools face a combinatorial blow up of the state-space, commonly known as the *state explosion problem*, that must be addressed to solve most real-world problems.

Building verification models for realistic applications that are both interesting and tractable does require significant expertise and time, and as a result model checking often is not push-botton technology. Nevertheless, for a large class of problems model checkers are extremely easy to use. When a group of high school students (age 15-16) visited our university a few years ago to learn about Computer Science, I asked them to analyze and correct a flawed design of a controller for a simple railroad crossing using UPPAAL. Without any training they discovered how to tackle the problem using the tool and at the end of the session some girls even asked where they could download this "cool" software package to continue playing with it at home. In a first-year mandatory course on Operating Systems this semester, after just one hour of training, CS students had no problem to use a model checker to validate their solution to the concurrency problem of Figure 2.[3] Among my students I did not observe any mathfobia

> Travellers come to a taxi stop and wait for a taxi. When the taxi arrives, all the waiting travellers invoke `boardTaxi`, but anyone who arrives while the taxi is boarding has to wait for the next taxi. The capacity of the taxi is 4 people; if there are more than 4 people waiting, some will have to wait for the next taxi. When all the waiting travellers have boarded, the taxi can invoke `depart`. If the taxi arrives when there are no travellers, it should depart immediately.
> The problem is to write synchronization code that enforces all of these constraints using semaphores, and to model and validate the correctness of your solution with the UPPAAL model checker.

Fig. 2. A simple concurrency problem.

or dislike of formal methods. They just appreciated that the tool helped them to solve their problem. Of course, applying model checking techniques on real industrial problems is somewhat more involved.

There have been numerous successful applications of model checking technology to industrial problems (see *e.g.* [13, 25, 27] for pointers). In terms of impact, the main application area is again validation of hardware circuits by companies such as Intel. But also in the field of network and communication protocols model checking has become an indispensible tool. Model checking has

[3] Models of semaphores were made available.

been applied successfully to all kinds of scheduling problems in manufacturing, transportation and real-time scheduling. Section 2.3 will describe some recent successes in model checking software. Within PROGRESS, the projects CES.5008, CES.5009, and TES.4999 have applied and further developed model checking technology. Below I report on some model checking case studies that were carried out by these projects.

One of the main applications studied by project CES.5008 was a system for lifting trucks (lorries, railway carriages, buses and other vehicles). This system consists of a number of lifts; each lift supports one wheel of the truck that is being lifted and has its own microcontroller. The controls of the different lifts are connected by means of a cyclic network. A special purpose protocol has been developed to let the lifts operate synchronously. When testing the implementation, the developers found three problems. They solved these problems by trial and error, partly because the causes of two of the three problems were unclear. In close collaboration with the developers at Add-Controls, the CES.5008 researchers modeled and analyzed the system in μCRL model checker [20]. The three known problems showed up in the model and in addition a fourth error was found. Solutions for all four problems were proposed and it was shown that, after incorporating these solutions, the model met all the requirements of the developers. The overall conclusion was that the μCRL model was an efficient tool to understand the behavior of this application. The case study also revealed limitations of the toolset and worked as a catalyst to have its capacities enlarged.

One of the case studies carried out by PROGRESS project TES.4999 (HaaST) was initiated by the home networking group of Philips Research. The study concerned the Zeroconf protocol[4], an IETF standard dedicated to the self-configuration of IPv4 network interfaces. The task was to investigate the trade-off between reliability and effectiveness of this protocol. The problem was tackled from different directions. The group in Twente analyzed a simple stochastic cost model of the protocol, where reliability is measured in terms of the probability to avoid an address collision after configuration, while effectiveness is viewed as the average penalty perceived by a user. The solution method was optimisation of several protocol parameters on minimal cost [5, 6]. The group in Nijmegen developed a UPPAAL model of the protocol in order to analyze functional correctness and real-time behavior [18]. The conclusion was that UPPAAL, which combines extended finite state machines, C-like syntax and concepts from timed automata theory, is able to model Zeroconf in a faithful and intuitive way, using notations that are familiar to protocol engineers. The modeling efforts revealed several errors (or at least ambiguities) in the Internet standard that no one else spotted before. Also a number of points were identified where UPPAAL still can be improved. After applying a number of (manual) abstractions, UPPAAL was able to fully explore the state space of an instance of the model with three hosts, and to establish some correctness properties.

Another case study carried out by project TES499, also proposed by Philips Research, concerned a distributed algorithm to monitor the availability of nodes

[4] See `www.zeroconf.org`.

in self-configuring networks. The simple scheme to regularly probe a node — "are you still there?" — may easily lead to over- or underloading. The essence of the algorithm is therefore to automatically adapt the probing frequency. It was shown that a self-adaptive scheme to control the probe load, originally proposed as an extension to the UPnPTM (Universal Plug and Play) standard, leads to an unfair treatment of nodes: some nodes probe fast while others almost starve. A very simple, alternative distributed algorithm was proposed that overcomes this problem and that tolerates highly dynamic network topology changes [7]. The analysis results have been obtained using the MODEST/MOBIUS tool suite. MODEST is a modeling language with a formal semantics [15] that has been developed within the project. The formality of the language allows not only for the integration with other formal analysis tools (such as model checkers), but, more importantly, is essential to carry out semantically sound simulation runs with MOBIUS. This results in a trustworthy analysis chain (one that can be validated by means of the semantics). Standard simulation environments are risky to use instead, because they have been found to exhibit contradictory results (both quantitatively and qualitatively, *i.e.* difference in behavior) even in simple case studies [12].

Model checking is applied very successfully and on a regular basis by verification experts in several niche areas. In many cases there can be no doubt that the technology is cost-effective. Nevertheless, much more effort is required before model checking will become main stream technology. The following problems need to be addressed:

1. *Scalability.* Model checkers must cope with the state space explosion problem. This growth often renders the mechanical verification of realistic models practically impossible: there just is not enough time or memory available. In order to make the models tractable, abstraction is required, but finding these abstractions can be a time consuming effort that requires expertise.
2. *Accessibility.* Building a good model is difficult because model checkers are mostly academic tools that lack extensive documentation and require a thorough knowledge of the underlying principles to build models that are suitable for analysis.[5] Thus, in practice, model checking tools are inaccessible to people with little or no background in formal verification.
3. *Relation between model and system.* The relationships between an (abstract) model of a system and the system itself is typically somewhat obscure. One can verify a high-level design, but what does that say about the realization of that design? As pointed out by Brinksma and Mader [10], current research seems to take the construction of verification models more or less for granted, although their development typically requires a coordinated integration of the experience, intuition and creativity of verification and domain experts. There is a great need for systematic methods for the construction of verification models to move on, and leave the current stage that can be characterized as that of "model hacking". The ad-hoc construction of verification

[5] This does not apply to some specialized in-house industrial tools that incorporate model checking techniques.

models obscures the relationship between models and the systems that they represent, and undermines the reliability and relevance of the verification results that are obtained.

4. *Convenience*. Model checkers usually are not a part of the development tool-chain with the result that there is little or no automation. Furthermore, many current tools and their input formalisms lack important features for convenient specifications in an industrial setting. As a result, modeling and analysis require a significant amount of time.

Much research is going on to extend the technology in these directions.

2.3 Automated Abstraction

As explained in the previous subsection, a key problem for model checking is scalability. Even though model checking technology has become very powerful, it is for instance typically not possible to fully explore UML models that are intended for code generation: when you try to do it these models just explode in your hands.[6] To check large systems, abstraction is therefore a key paradigm: the purpose of an abstract model is to retain those features of a system that are necessary to verify the desired property, and to omit all unnecessary detail.

For verification of hardware and manually constructed models of embedded systems, many generic abstractions (*e.g.* symmetries, data-path, abstract interpretation) have been proven useful. Within PROGRESS projects CES.5009 and TES.4999 powerful abstraction techniques have been added to the model checking tools μCRL and UPPAAL, abstract interpretation and symmetry reduction, respectively, thereby greatly enhancing their applicability.

An even more ambitious approach has been followed by the SAL (Symbolic Analysis Laboratory) project at SRI [2]. SAL is a framework for combining different tools to calculate properties of concurrent/reactive systems. The heart of SAL is a language for specifying concurrent systems in a compositional way. The current implementation of the SAL framework augments PVS with tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking to separate concerns as well as calculate properties (*i.e.* perform symbolic analysis) of concurrent systems. Altough it is still in the prototype stage and its usefulness for tackling industrial problems needs to be demonstrated, SAL can be viewed as a promissing attempt to bridge the gap between model checking and theorem proving.

Recently a number of breakthroughs have been achieved and we see, for instance, that model-checking techniques are now being applied to validation of source-code (in particular C and JAVA) — so-called *software validation* or *run-time verification*. Noticeable successes in this area have been obtained by the SLAM and Blast projects and tools. A basic technique used by these tools is *abstraction-refinement*. In abstraction refinement an initial very course abstraction of a program is computed automatically. In this abstraction, for instance,

[6] This visual description is due to Koos Rooda.

the only information about an integer variable that is preserved is whether it is zero, positive or negative. Or, alternatively, all valuations of program variables that cannot be distinguished by any of the Boolean guards that occur in the program are deemed equivalent. Next exhaustive state space search (model checking) is used to explore the abstract model. If in the abstract model no "bad" state can be reached then we know by construction that no bad state can be reached by the original program. In this case we have established correctness of the program, and we are done. In case a bad state can be reached in the abstract model then there are two possibilities:

1. either there is a corresponding execution of the original program that leads to a bad state; this means that we have found a bug in the original program,
2. or the bad execution in the abstract model does not correspond to any execution in the original program; in this case we can use the information about the failed correspondence to construct a refinement of the abstraction, that is, a new abstraction that is in between the old abstraction and the program, and we repeat the analysis.

SLAM and Blast have been succesfully applied within the domain of debugging of device drivers (programs with over 100,000 lines of C code). In his keynote address at WinHec 2002, Bill Gates referred to the SLAM project as follows:

> *"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability."*

I expect eventually it will be possible to apply software model checking also to the analysis of embedded software and to UML like models. Still, scalability remains a key issue and in order to enable routine use of formal verification techniques in the embedded systems area much further research is needed.

2.4 Invisible Formal Methods

Even though manual construction of abstract verification models can be very rewarding and helps to obtain insight and improve a design, practitioners of course prefer to have push-botton verification technology that can be applied directly to their UML models and software. The concept of *types* and the development of automatic algorithms for establishing type correctness is one of the big successes of formal methods research. The algorithms and their underlying math are completely invisible to the user, but still the return on investment is excellent. Model-based development provides the artifacts needed by automated analysis, and this creates some exciting new opportunities for applying mathematical analysis techniques. Commercial tools such as Rational Rose (Real-Time), Rhapsody and visualSTATE support verification of certain functional correctness properties (*e.g.* absence of deadlock). Hidden from the engineer sometimes very sophisticated formal methods are being used to provide these results. For

these invisible formal methods, convenience is more important than generality. They will not find all the bugs in your design but they will find most of them fast and automatically.

Formal Verification A nice example of invisible formal methods is provided by visualSTATE, a suite of graphical tools for design of embedded systems and event-driven systems developed by IAR Systems. The tool uses a sophisticated verification algorithm called compositional backward reachability analysis to exhaustively verify large industrial applications—comprising more than 1,000 components—in a few minutes on a standard PC [34]. This (patented) technique allows designers to test that their state machine design model and embedded application does not contain any of the following problematic properties:

- state, local and system wide deadlock conditions,
- conflicting transitions between states,
- unreachable states, *i.e.* states that cannot be entered by any sequence of events from the environment,
- unused events or signals, *i.e.* stimuli to the system that is not acted upon,
- unused transitions, *i.e.* transitions that will never fire, regardless of the event sequence fed into the system,
- unused actions or assignments,
- unused variables, parameters and constants.

In the realms of software, analysis tools such as the Extended Static Checker for Java (ESC/Java, [17]), turn out to be very effective. ESC/Java is a programming tool that attempts to find common run-time errors in JML-annotated Java programs (*i.e.* Level 2 interface specs according to the classification of Beugnard et al [3]) by static analysis of the program code and its formal annotations. Users can control the amount and kinds of checking that ESC/Java performs by annotating their programs with specially formatted comments called pragmas. Because ESC/Java abstracts from the full Java semantics it will not spot all the program bugs that analysis with a theorem prover such as the LOOP tool will reveal. But because it is automatic, ESC/Java is in most cases much more effective.

Still the type of properties that can be verified using invisible formal methods is restricted. From the point of view of embedded systems, MDD tools have a serious lack of support for predicting real-time behaviour, resource-consumption and performance in general of the generated code (Level 4 properties). Clearly, much more research is needed in this direction. Given the effectiveness of invisible verification techniques (see also Figure 1), I consider this to be an important research direction.

Correctness of Implementations Bridging the gap between high-level modelling or programming abstractions, and implementation platforms is one of the key challenges for embedded software research [33, 26]. Tools such as Rational

Rose (Real-Time), Rhapsody and visualSTATE allow us to generate code directly from models, but how do we know that this code is actually correct? In particular, how do we know that the generated code meets hard real-time constraints?

An important step towards supporting quantitative analysis of real-time aspects is provided by the modelling formalism of timed automata. Since their introduction by Alur and Dill [1] in 1990, several verification tools for timed automata have been developed, which are now applied routinely to industrial-size case studies. However, as yet, there is no support for generation of *predictable* code from timed automata models. In fact, this is a nontrivial research problem due to the arbitrary precision of clocks in timed automata.

The problem of providing a *predictable design trajectory* has been discussed at length by Henk Corporaal in his white paper [14] and his group has made some important first steps towards a solution [22]. Fully solving this problem will require extensive use of formal methods, which in the end will be invisible to the designer.

Testing Model Based Testing (MBT) aims at the automatic creation, execution and evaluation of test cases to test software systems. In most software projects testing is done by hand. There are some tools that automate parts of the test process, like test execution and/or test evaluation. The goal of MBT is to automate the entire test process. The claimed benefits are:

- Better coverage of functionality. MBT can create and execute more and better test cases than humans can. MBT is very thorough, in principal it can cover the entire functionality of the system. It can for instance generate test vectors that will drive an implementation through all the states and transitions of its model.
- Faster testing. Everything is automated and as a result we can test faster. This is especially important in the test execution fase, as this fase is close to the delivery deadline and is under a lot of time pressure.
- Cheaper testing. MBT enables more thorough testing with less people (in less time)

MBT uses a model of the system that is under test (so called SUT: System Under Test). The model describes (part of) the behavior of the SUT (functional and/or extra-functional behavior, like timing, performance, *etc.*).

For testing of *control-dominated systems* (*i.e.* systems with a high degree of interaction with their environments) there is a rich and well-understood theory of model-based testing (part of which was developed within the PROGRESS project TES.5417), which has been (partially) implemented in a number of model-based testing tools such as:

- The Reactis Simulink Tester generates test suites automatically from Simulink or Stateflow diagrams. Each test consists of a sequence of stimulus/response pairs, where each stimulus assigns an input value to each in-port in the model

and each response records an output value for each out-port. The test suites are generated from a coverage criteria of the specification, *e.g.*, transition or state coverage.
- The Conformiq Test Generator automatically generates test cases from UML state chart models. Simulations of the models can be used to generate batches of test cases that can later be executed. Alternatively, the models can be interpreted dynamically to facilitate on-the-fly testing.
- Similarly, the Statemate MAGNUM ATG (I-Logix) tool uses model-checking and simulation techniques to derive test sequences from state chart models.
- RT-Tester (Bremen) and TorX (University of Twente) are both tools with an underlying formal theory and are rooted in academia. Both tools are for on-the-fly test generation and execution, where the specification is continually probed for relevant input stimuli and used to check the validity of output actions. RT-tester accepts specifications in a mixture of languages, but mainly timed CSP, whereas TorX accepts Promela or LOTOS.
- TGV (Irisa) and Telelogic TestComposer are SDL based test case generators. Given an SDL specification and a test purpose (or a specification coverage criterion) these tools construct a test case that meets the test purpose, and stores this in TTCN format. Phact (Philips Research) TestGen (INT, France) also produce TTCN test suites, but uses FSM checking experiment based test generation.

Tools like TorX and TGV allow for the on-line and off-line generation of sound and complete test suites from discrete, state-based models, such as *Labeled Transition Systems*, and they can use *test purposes* to steer the test derivation algorithms to explore behaviours of the SUT that are more likely to contain large amounts of bugs. For many systems, however, such simple state-based models are not sufficient. They require richer models, which include quantitative information, such as real-time, continuous or complex data variables, and stochastic properties (*e.g.* performance, statistics, probabilities). Although some early prototype tools exist that combine control with time, data, or stochastics, *e.g.* STG, TTG, UPPAAL Tron, and an extension of TorX, the theory regarding model-based testing for quantitative models is still in its infancy.

Although there can be no doubt that MBT is a very important and interesting technique, which eventually will find its way into all major MDD tools, the cost-effectiveness of current MBT techniques is not evident. Different authors arrive at different conclusions. Campbell et al [11] report enthousiastically on a MBT tool Spec Explorer, which is being used daily by several Microsoft product groups. In one particular setting, their model-based approach helped to discover 10 times more errors than traditional test automation and the kind of bugs discovered were deep system-level bugs (*i.e.* bugs that were only found after the system performed many steps), for which manual test cases would have been hard to construct. This story is in sharp contrast with a report by Pretchner [30] (presented at the very same meeting), who took a critical look at MBT and concludes that, to the best of his knowledge, there is no published evidence that the promises of MBT are kept. Although the study by Pretchner is too small

to make generic conclusions, it is clear from his experiments that the benefits of MBT should not be taken for granted. A lot of further research will be required on MBT to turn it into a mature technology. A challenging question, for instance, is to find good measures for coverage.

Using Models for Diagnosis and Control As human beings we maintain numerous models about ourselves and the world we live in. We use these models to interpret our observations of reality, to analyse causes when something goes wrong, to predict the future, and to device strategies for well-being and survival. In this light it is very natural that the models of computer based systems that we construct with MDD tools are not only used to generate software and to predict the future (which is essentially what verification and validation is about) but also for interpreting observations of implementations and as a basis for controlling physical systems.

Within control theory, model predictive control (also called model based control) is an industry proven solution to complex process control problems that started out in the late 1970s in the refining and petrochemical industries. At the heart of MPC is a mathematical model of the process that is used to predict future process behaviour. Using this predictive model the controller is able to calculate an optimum set of process actuator moves which minimise the error between actual and desired process behaviour subject to actuator and process constraints.

PROGRESS project DES.7015 uses a model-based approach for fault diagnosis. If a system does not behave according to its specification, what is the root cause of this failure, and what can we do about it? Solving this problem requires sophisticated probabilistic reasoning.

3 PROGRESS Projects on Verification and Validation

In the previous section, several results obtained by PROGRESS projects have already been described. In this section, I will briefly summarize the goals, results and utilisation for each of the six PROGRESS projects in the verification and validation area. For more information I refer to the project websites, which are accessible via `http://www.stw.nl/programmas/progress/`.

3.1 CES.5009: Real-time Distributed Shared Data Space

Goals The main goal of this project was to evaluate the applicability of the μCRL language and tools on some large-scale industrial applications, and to improve this verification technology where needed. More specifically, the goals of the project were (1) to develop shared data space architectures, (2) to formally model such architectures, (3) to verify software applications based upon such architectures, and (4) to develop the verification technology needed to scale to shared dataspace applications.

Results The following results were obtained:

1. A model for describing systems based on heterogeneous shared dataspace paradigms. Special instances of the generic model are (the essential core of) Splice (Thales), 4TEC (4TEC), JavaSpaces (SUN Microsystems). A methodology has been developed and implemented to automatically obtain distributed prototype implementations from such models. Also, a methodology has been developed and implemented to automatically verify the models.
2. A formal model of the JavaSpace architecture. This model includes all relevant features: reading/writing, transactions, notification, leasing, and timeouts. As a result, JavaSpace program models can now be automatically verified by a model checker.
3. A line of tools for distributed model checking. This allows us to scale verification methodology by using clusters of PCs. These tools are now part of the muCRL tool set.
4. Tools for applying abstract interpretation. Given a default or user-specified abstraction, the tool automatically generates a smaller state space. The abstraction tools have been implemented and integrated in the muCRL toolset.

Utilisation The analysis of shared data space architectures revealed problems, provided solutions, and brought a lot of methodology and background knowledge to both companies involved in the project. The solutions provided by the CWI researchers found their way in subsequent versions of Splice and 4TEC. Both tools that were developed have been applied in many other projects, including PROGRESS project CES.5008.

3.2 TES.4999: HaaST: Verification of Hard and Softly Timed Systems

Goals The HaaST project aimed at the development and integration of methods and tools for the verification and analysis of real-time embedded systems, with an emphasis on distributed algorithms and protocols for consumer electronics applications. The goal was not only to consider "hard" real-time constraints – those that require that a system must react in time – but also so-called "soft" real-time constraints – those that require that the system should react in time but occasionally may not.

Results Within HaaST a prototype tool MOTOR has been developed for model checking stochastic systems. The project also contributed to the further development of the timed model checker Uppaal, a tool that is now being used by thousand of researchers both in academia and industry. Verification of hard and softly timed systems is considered as a most important topic by the international research community, with great societal relevance, and many strong groups are working on it. It is evident that during the lifetime of the project enormous progress has been made in this area, with HaaST active on the front line. With contributions from the HaaST project, the model checker Uppaal has advanced

from an academic proof of concept to a tool that is being downloaded by thousands of researchers both in academia and in industry, and that is now ready for further industrial development. The MOTOR tool, which is a true product of the HaaST project, is still very much in the stage of an academic prototype, but its potential usefulness has been demonstrated already on some industrial case studies, and clearly further development of this tool will be most promising. Altogether, the results of the HaaST project and the case studies that were carried out indicate that in many cases it can be advantageous (and cost effective!) to perform formal modelling and analysis of (timing related or other) properties of embedded systems using these methods. The precise modeling of the system under consideration often already gives an important benefit, and, the combined assessment of both quantitative and qualitative system requirements using the same system model can be of great value.

Utilisation The results of the HaaST project are being used in several European research consortia on embedded systems with strong industrial participation, notably the IST-project *Advanced methods for timed systems (AMETIST)* and the network of excellence ARTIST. Also at the national level there are currently several research projects that are strongly related to HaaST and that are (partially) based on its results. The model checker UPPAAL is by thousands of researchers both in academia and in industry. See `http://www.uppaal.com/` for an incomplete listing of industrial application.

3.3 CES.5008: Improving the Quality of Embedded Systems Using Formal Design Techniques

Goal To formally model and analyze some embedded systems that were under development by the company Add-Controls.

Results Besides the succesful modeling and analysis of the system for lifting trucks that was described in the previous section, also embedded controllers for a staircase elevator and a hydrolic cylinder have been analyzed. Also a cache coherence protocol that was designed by the parallel systems group at the Free University has been analyzed, and again a flaw in the design was found [28]. Quite a number of other "academic" algorithms and protocols were successfully analyzed using a wide variety of tools: μCRL, CADP, PVS, UPPAAL and PRISM.

Utilisation After completion of the project, Add-Controls remained interested in the use of model checking technology for analyzing its designs. Under supervision of Wan Fokkink and Jun Pang, an MsC student from the Radboud University Nijmegen analyzed a redesign from the lift system using UPPAAL.

3.4 EES.5141: Specification Tooling for Embedded Software Components

Goals Component technologies such as DCOM, CORBA and Java Beans are being used in an increasing number of industrial embedded systems. In com-

ponent technology, interfaces play a key role; one component can have more interfaces. Components deliver and use services through explicit interfaces only. Proper interface specifications are a prerequisite in assuring the interoperability of components within a system. The goal of this project was to take a mainstream extensible CASE tool supporting UML-based object-oriented modelling techniques (*e.g.* Rational Rose), and customize it in such a way that it provides optimal support for developing and deploying interface specifications for embedded software components. The customization relied on the ISpec interface specification methodology developed and used at Philips [23]. The formal underpinning of this template-based methodology, involving semantics for UML, ISpec and plug-in component decriptions for ISpec templates, was one of the challenges of this project.

Results A semantics and a tool have been developed [32]. The latest version of the tool, which has been named Calisto, is available at `www.win.tue.nl/calisto`. The Arkas Software Engineering Student Working Group has been involved in the further development of the tool: it has been converted to the .NET framework, suitable for use with Visio2003, and well documented.

Utilisation The tool is being used at the LaQuSo and within ISpec courses within Philips. The results and ideas generated by the project have also been used within ITEA-DESS and ITEA-EMPRESS. Possible applications at Philips Semiconductors, Philips Medical and Oce are being explored. There are also contacts with ASML.

3.5 TES.5417: Atomyste: Atom Splitting in Embedded Systems Testing

Goal The goal of Atomyste has been to enable changes in the model and in test cases. Atomyste focusses on specific type of changes, namely "action refinement". Action refinement means that we take an (incorrect) action in the model and replace it with correct or better behavior. For example, suppose our model tells us that we can enter a one euro coin in the SUT and we find out that the machine also accepts two fifty cent coins. Action refinement enables us to replace the "one euro" behavior with behavior that also allows two fifty euro cent coins. As a result we can change the model and then (automatically) make new test cases that reflect the change, or we can directly change already existing test cases.

Results Atomyste extended the MBT theory to enable action refinement in MBT and implemented the new theory in a prototype test tool. The effort that it takes to create and maintain a model is important for the succes of MBT. Hence the results of Atomyste are important for MBT.

Utilisation Mainly through the ESI project TANGRAM, which has ASML as carrying industrial partner.

3.6 DES.7015: FINESSE: Fault Diagnosis for Embedded Systems Dependability

Goals The ability to accurately diagnose and recover from faults in complex systems such as the copiers of Oce constitutes a crucial element in achieving higher system dependability. As effective recovery (or repair) fully depends on the accuracy of the fault diagnostic process to determine the root cause of failure, fault diagnosis (FD) is the key determining factor. Apart from the operational phase, FD is also beneficial in the development phase where many system faults occur as a result of improper design and/or integration.

The FINESSE project develops and investigates an improved FD strategy, based on a novel FD method within a model-based approach. The method provides the required diagnostic accuracy to meet the challenges posed by the complex application carrier. The model-based approach reduces the embedded FD software development effort since it is also used to generate code. As the model-based approach is relatively well-established, the FD method is the central theme in FINESSE.

Diagnostic models of complex systems usually allow for many diagnostic solutions, ordered in terms of probability, while only one of the solutions reflects the actual system health (*e.g.* the combination of HW component X and SW component Y is unambiguously at fault). In order to radically improve FD accuracy compared to the current state-of-the-art, the project proposes to (1) improve the quality of the probabilistic diagnosis ranking process, and (2) to significantly decrease the number of diagnosis solutions. To address the former, an improved fault probability modeling method is developed to estimate the a priori probability of faults occurring in software components, which is much more complex than hardware component fault probability modeling. To address the latter, an improved FD algorithm is developed which includes the ability to reason over time at low-cost as well as to automatically generate test vectors as part of the diagnostic reasoning process.

The FD approach will be implemented in terms of an existing, model-based tool set based on TUD's system modeling language Lydia, and validated on a paper handling system (PHS) of Oce in terms of a demonstrator. The issues that will be investigated include the adequacy of the new FD approach to improve system dependability during operations, the effort spent in modeling, the computational costs of the FD approach, all compared to traditional techniques, as well as architectural development topics such as the added (dependability) value of improved sensor placement, and improved testability features.

Results No result yet: the project just started.

Utilization Apart from Oce and LogicaCMG, the impact of the research is expected to be very high. Many manufacturers that produce complex hardware-software artifacts performing functions with a high economic added value and/or which are life-critical, are facing tremendous problems with respect to systems dependability, and have traditionally spent a huge effort on devising FD mechanisms. On a national scale examples can be found at industries such as ASML,

Philips (Medical Systems, Consumer Electronics, Semiconductors), apart from Oce.

4 Conclusions

Boosted by the advent of UML and MDA, the role of models during the design, implementation, verification and validation of embedded systems has become much more important in recent years. This is a very positive development which indicates that very slowly embedded system design is becoming a mature engineering discipline. Commercial tools for model driven development, such as Rational Rose, Rhapsody and visualSTATE, have gained popularity primarily because they support automatic code generation from abstract models (various variants of StateChart). By developing systems at a more abstract level that is (more or less) independent of the specific hardware platform, reuse becomes possible and this saves money.

Model driven development provides a great opportunity to improve the verification and validation process through the introduction of formal techniques. The systematic, structured construction of models by itself already supports validation and verification. In addition, the fact that models are available in machine readable form enables the application of a whole range of (mathematical) techniques for analysis such as theorem proving, model checking, model based testing and runtime verification. These techniques are still far from main stream technology at the moment, but play an increasingly important role in certain niche areas [27]. Their economic value is certainly demonstrated in those cases.

A general cost/benefit analysis and comparison to other approaches are seldom made or are at best very limited in scope [25]. One of the problems is that the whole area is subject to so much change: by the time you have made a cost/benefit analysis it is outdated. We have *e.g.* seen enormous progress on formal verification tools during recent years: scalability, accessibility, convenience and realizability have all been drastically improved. Another issue was raised by Gerrit Muller at a recent ForTIA meeting [27]. He conjectured that (at least at the system design level) the added value of formal methods are primarily the skills of the people using them: they are analytical, structured, firm in principle and consistent. Using these skills, these individuals can play an important role in an informal multi-disciplinary process, but not necessarily using mathematical models or applying rigorous analysis techniques. I believe that Muller's conjecture is wrong (and hope this paper has provided enough evidence for this): making formal models is a great way of finding ambiguities/mistakes in designs, and symbolic calculation/search by formal verification and validation tools also helps to find many more nontrivial bugs. Nevertheless, if one tries to make an objective assessment of the benefits of formal verification and validation methods, the issue raised by Muller is of course very relevant.

Formal verification is one of the tool boxes that can (and sometimes has to) be used in the construction of embedded systems. In many situations application of formal verification is not (yet) cost-effective, but in many other situations it

does pay off. The challenge is to recognize these situations. Right now within most Dutch companies in the embedded systems area there is not too much knowledge about formal verification technology, although certainly initiatives such as PROGRESS and ESI have helped to improve things. Still, most companies do not realize how important it is for their own success to be *experts* on V&V. It is essential to have at least a rough feeling concerning what formal verification and validation can and cannot do. Within Dutch Universities there is much expertise on formal methods and either directly or through organizations such the Laboratory for Quality Software (LaQuSo, `http://www.laquso.com`) of the Universities of Eindhoven and Nijmegen, this expertise can be easily used. When I completed my PhD thesis in 1990, the typical duration for a formal verification case study was one year. Due to advances in the field it is now often possible to get the first results after one week (depending on the case study, of course). Big companies such as Intel, Siemens, IBM Lucent and Microsoft have dedicated groups working on the development and application of formal verification technology. Within the Netherlands verification specialists are active within *e.g.* Philips, Imtech and Chess. My impression is that Pentium bug style disasters will be needed to convince Dutch companies to set up full-fledged formal verification groups. Maybe the situation will change within a few years due to increased use of model driven development, the fact that formal verification technology becomes more and more powerful, and the integration of model driven development with formal verification technology.

There appears to be a big difference in mental attitude in the hardware and in the software community. In the former, the use of formal techniques is well established, possibly because product liability claims are of real economic significance. In the software community, product liability is typically waived and the end-users still seem to accept that fact. Quite likely, the uptake of formal methods in main stream software engineering is hindered by that. There is evidence that in the area of embedded software, where the borderline between hard- and software is inherently less obvious, this attitude is in fact changing [27]. The quality demands posed on those type of systems, for example in the automotive domain, are typically identical to hardware, and product liability is indeed a real concern here, which raises the need for system verification and validation.

Acknowledgements Thanks to the PROGRESS organizers for inviting me to write this overview. I apologize for the fact that writing it took more time than expected! Thanks to Jaco vd Pol, Ruurd Kuiper, Erik Poll and Marcel Verhoef for comments on a draft version.

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL.

In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

3. A. Beugnard, J. Jézéquel, and N. Plouzeau. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.
4. B.W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
5. H. Bohnenkamp, H. Hermanns, M. Zhang, and F.W. Vaandrager. Cost-optimisation of the IPv4 zeroconf protocol. In *Proceedings of the 3rd PROGRESS Workshop on Embedded Systems,* Utrecht, the Netherlands. PROGRESS/STW, 2002. ISBN 90-73461-34-0.
6. H. Bohnenkamp, P. van der Stok, H. Hermanss, and F.W. Vaandrager. Cost-optimisation of the IPv4 zeroconf protocol. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2003)*, pages 531–540, Los Alamitos, California, 2003. IEEE Computer Society.
7. H.C. Bohnenkamp, J. Gorter, J. Guidi, and J.-P. Katoen. Are you still there? - a lightweight algorithm to monitor node presence in self-configuring networks. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*, pages 704–709. IEEE Computer Society, 2005.
8. B. Bouyssounouse and J. Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
9. J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods ...ten years later. *IEEE Computer*, 39(1):40–48, 2006.
10. E. Brinksma and A. Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, January 2004.
11. Campbell C, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In J. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547. Springer, 2005.
12. D. Cavin, Y. Sasson, and A. Schiper. On the accuracy of manet simulators. In *Proceedings of the 2002 Workshop on Principles of Mobile Computing, POMC 2002, October 30-31, 2002, Toulouse, France*, pages 38–43. ACM, 2002.
13. E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.
14. H. Corporaal. Embedded system design, 2006. Progress Mini Symposium White Paper.
15. P.R. D'Argenio, H. Hermanns, J.-P. Katoen, and R. Klaren. Modest - a modelling and description language for stochastic timed systems. In L. de Alfaro and S. Gilmore, editors, *Process Algebra and Probabilistic Methods, Performance Modeling and Verification: Joint International Workshop, PAPM-PROBMIV 2001, Aachen, Germany, September 12-14, 2001, Proceedings*, volume 2165 of *Lecture Notes in Computer Science*, pages 87–104. Springer, 2001.
16. S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1987.

17. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.
18. B. Gebremichael, F.W. Vaandrager, and M. Zhang. Analysis of a protocol for dynamic configuration of IPv4 link local addresses using Uppaal. Report ICIS-R06016, Institute for Computing and Information Sciences, Radboud University Nijmegen, 2006.
19. B. Gebremichael, F.W. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu. Deadlock prevention in the æthereal protocol. In D. Borrione and W.J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 345–348. Springer, 2005.
20. J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *J. Log. Algebr. Program.*, 55(1-2):21–56, 2003.
21. L. Heerink and E. Brinksma. Validation in context. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, volume 38 of *IFIP Conference Proceedings*, pages 221–236. Chapman & Hall, 1996.
22. J. Huang, J.P.M. Voeten, O. Florescu, P.H.A. van der Putten, and H. Corporaal. Predictability in real-time system development. In *Advances in Design and Specification Languages for SoCs*, Dordrecht (The Netherlands), 2005. Kluwer Academic Publishers.
23. H.B.M. Jonkers. Ispec: Towards practical and sound interface specifications. In W. Grieskamp, Th. Santen, and B. Stoddart, editors, *Integrated Formal Methods, Second International Conference, IFM 2000, Dagstuhl Castle, Germany, November 1-3, 2000, Proceedings*, volume 1945 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2000.
24. H.B.M. Jonkers. Interface specification: A balancing act (extended abstract). In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, volume 3054 of *Lecture Notes in Computer Science*, pages 5–6. Springer, 2004.
25. D.R. Kuhn, R. Chandramouli, and R.W. Butler. Cost effective use of formal methods in verification and validation, 2002. Paper presented at Workshop on Foundations for Modeling and Simulation (M&S) Verification and Validation (V&V) in the 21st Century (Foundations 02), October 22-24, 2002, Johns Hopkins University Applied Physics Laboratory, Laurel, Maryland (USA).
26. E.A. Lee. What's ahead for embedded software? *IEEE Computer*, 33(9):18–26, 2000.
27. T. Margaria, B. Schätz, and M. Verhoef. Formal methods going mainstream — cost, benefits and experiences, 2006. Report on the ForTIA Industry Day at FM 2005.
28. J. Pang, W. Fokkink, R.F.H. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 238. IEEE Computer Society, 2003.
29. K.R. Popper. *Conjectures and Refutations*. Routledge and Kegan Paul, 1963.

30. A. Pretschner. Model-based testing in practice. In J. Fitzgerald, I.J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 537–541. Springer, 2005.
31. P.E. Ross. The exterminators. *IEEE Spectrum*, pages 36–41, September 2005.
32. E.E. Roubtsova, L.C.M. van Gool, R. Kuiper, and H.B.M. Jonkers. Consistent specification of interface suites in uml. *Software and System Modeling*, 1(2):98–112, 2002.
33. S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill. Scanning the issue - special issue on modeling and design of embedded software. *Proceedings of the IEEE*, 91(1):3–10, 2003.
34. J. Staunstrup, H.R. Andersen, H. Hulgaard, J. Lind-Nielsen, K.G. Larsen, G. Behrmann, K.J. Kristoffersen, A. Skou, H. Leerberg, and N.B. Theilgaard. Practical verification of embedded software. *IEEE Computer*, 33(5):68–75, 2000.